CSC442 Artificial Intelligence
Project 1

Andrii Osipa
URID: aosipa

September 2017

1 Part A

1.1 Notation, data types, etc

Def. We will call a move "best" if it is one with highest minimax utility in the current game state. Obviously, there may be multiple "best" moves.

Game state is represented as a 10-element array, where first 9 elements are field itself in the left to right row by row order. Values of cells are from $\{-1, 0, 1\}$, where -1 corresponds to an empty cell, 0 to one with O and 1 to the one with X in it. The last element of an array is the result of the game. It is from a set $\{-2, -1, 0, 1\}$, where -2 means that game is not over, -1 means O wins, 0 means draw, and 1 means X wins.

Everything else about game state may be easily computed from the information already stored. Obvious, that if $(\sum_{i=1}^{9} field[i])\ mod\ 3)$ is equal to 0 then it is X's turn to do a move. Otherwise, it's O's turn.

As utility function does not depend on who player(an entity that gives input to the algorithm) plays for, therefore the player is not always considered max in minimax algorithm, but they are maxed when they play for X and min when they play for O.

Tree structure: node contains only its current game state, possible next game states, and list of minimax utility for the next possible game states. There is no connection to the parent of the node as it is never used. Also, everything before the current game state is not available to access. Anyway, it's useless.

1.2 General algorithm

As this problem does not have big state space we can explore tree completely and do not need any heuristic. The algorithm does exactly this, except some branches are not explored initially, exploration continues until one of the two: find a perfect path(steps have max/min possible utility), every node in the tree is explored.

Notation: we will call a move precomputed if the tree was explored enough to find out minimax utility for this move. Each move will be computed only once during the whole algorithm.

- **Initial tree search:** DFS to explore possible moves, which are not explored before. Starts from some node with some game state returns minimax utility of the game state it started from. If at some moment of time it found the best move according to minimax strategy(which leads to winning of the party, whose turn it is), then no more search is concluded at that level. So not everything is explored in this function and it is not a complete DFS. This will affect when other party takes not the best moves or in cases when there are multiple best moves for the party and it uses the one, which was not precomputed by the algorithm. In those cases, some additional exploration of the tree may be required.

- **Possible move search:** takes node with some game state and returns best possible move at this moment of time. Short algorithm description:

```
if there is move with best minimax utility in precomputed moves:
    return this move;
else:
    if all possible moves are precomputed:
        return best from the set of precomputed moves;
    else:
        explore all possible moves;
            (Comment:now all of them are in precomputed)
        return best from the set of precomputed moves;
```

Initially, algorithm runs Possible move search, which itself will run DFS itself if there is a need to do so. There is a chance that all the game will go accordingly to the path computed by DFS at the first step of the game. In this, there will be only one run of DFS during the whole game.

1.3 Optimization

Used the following pruning technic: if during the computation of minimax utility of the possible moves we achieved best possible value (1 for max and -1 for the min) then we do not explore any other possible moves. The other thing is that before running DFS algorithm will check if it has the best move already.

Another point is the following: nothing is computed more than once. Generally, everything that can be reused in the future may be reused if needed.

1.4 Results

Always get a draw when I play with the computer if my goal is to win. Draw when plays with itself. Times for moves: the first move for X is done in 0.9-1s; the first move for O is done in 0.036s. Other moves are done in around 10^{-4}s. There is an obvious way to improve first time by a factor of three: there is no point to check all 9 moves for X as there are only 3 unique moves and every other is just a rotation of the field.

1.5 Instructions to run

Just run command "python aosipa_a.py". Everything else will be prompted as needed. Input and output are done according to the requirements in the document. For input sys.stdin is used, other messages are in sys.stderr, computer moves are in sys.stdout.

2 Part B

2.1 Algorithm

2.1.1 Structure

Game state here is a class that contains the following parameters: board(9x9 array), list of indexes of boards, where next move is allowed, result of the game(int). Each tree node contains game state, it's α, β parameters, list of nodes, that are its children.

During operation, the following statistics are collected: time of the search and number of cut branches during pruning.

2.1.2 General notes

The implemented algorithm is limited depth DFS. As with practically reasonable depth of the search, we will not often encounter terminal nodes, so almost everything will be evaluated by heuristics function. As heuristics is not admissible and generally, it can overestimate and underestimate states easily, so at each step of the game new search tree is computed from starting position with given depth. At this point this algorithm is very different from the previous algorithm in Part A as there we had an opportunity to reuse tree computed before.

There is also implemented $\alpha, \beta-$pruning. Therefore each step done by machine is the one that maximizes(minimizes) worst possible result.

Note: to get better results of pruning and get more interesting games all moves are shuffled after generated. For pruning this makes sense as a random order of succeeding will give us a good improvement on average as stated in AIMA. And interesting games is also important, as otherwise all moves of the machine will be strictly defined by the moves of another party.

Note: move selection is done by the rule stated above, but! sometimes move may not be optimal in terms of game length, despite the result will be the best. To be more precise, the algorithm does not give preference to terminal moves. Moves are randomly shuffled and first with the best possible score is selected. What happens in this case: sometimes there is a possibility to end game in 1 move and a possibility to end it with 3(5,...) moves. The algorithm will not give preference to the one, that ends game faster as this is not set up to be a goal.

2.1.3 Depth of the search

I was experimenting with the depth of the search to find highest value that does not gives high move times. The resulting depth is the following function over time(game step #): $depth(n) = \begin{cases} 0, n = 0 \\ 3, 1 \leq n \leq 30 \\ 4, otherwise \end{cases}$ Reasoning is the following: if we want to explore tree on 0^{th} step it means algorithm plays for 0. After playing a couple of games, I assume the first move of X does not matter. Therefore $depth(0)$ is limited to 1. Another important reason for this is that this exploration is generally the largest one, as there is everything possible and the first move has 81 possible options. Even with $depth(0) = 3$ time for the first move is significantly larger than other moves.

The reasoning for increased depth after 30^{th} step: the first reason is that I can afford it. Deeper search is better search, so it is definitely positive change. Why we can do it: increase in the move time is not significant, which was proved by running many games. An example is given is the section about optimization. From it, we can see that times for first two moves are the highest ones and they still stay below $0.2s$. After that, there is a decrease(not monotonic).

Why does that make sense to improve search on the 30^{th} move? It seems that game is almost over at this moment of time, but that is not true. Recorded games with over 40 moves. Therefore on 30^{th} move even with depth 4 end of the game, probably, will not be discoverable.

Position 30 was found just by experimenting and keeping track of move times(basically, we want as small position as possible).

2.2 Heuristics

During my work on this project I tried a few heuristics and here is the list:

- Giving 0 scores for everything. This one is kind of useless as with very small exploration depth, every leaf at the currently explored tree will have score 0 as chances to get to the terminal states is very very low and therefore all preceding nodes will have 0 by minimax strategy.

- Let's say that board is controlled by X(O) if X(O) can win it with one move. Basically, this means there is a row/column/diagonal, where two cells are Xs(Os) and the third one is empty. Now score for the game state(set of 9 boards) will be computed in the following way: We add

0.1 to the score for each board, controlled by X and subtract 0.1 for each board controlled by O. Easy to see that score is in the interval $[-0.9; 0.9]$.

Argumentation for this scoring function: in this game, it is very important to "control" boards to limit moves of the opponent and have a chance to win. And it's valuable to keep other party control as little as possible boards as this gives more opportunities for moves(that do not lead directly to failure). This scoring increases for each player when they take control over a board or force another player to lose control over a board.

This heuristic is what is used in my algorithm.

- Next heuristic is somewhat similar to the previous one, except now we say the board is controlled by X(O) in k ways if X(O) can win it by doing k different moves. For each way board is controlled by X we add $\frac{1}{60}$ to the score and subtract if it is controlled by O.

  This heuristics is not bad, but it's a disadvantage that with it taking control over new board results in the same score as getting one more way of control over a board, that's already controlled by the party. And it's better to get control over a new board, generally.

Note: these heuristics are only applied to non-terminal states. If the state is terminal it is scored correspondingly. Score for the state where X wins is 1 and where O wins is -1. All these heuristics give strictly smaller absolute value, so they will not give a bigger score to non-terminal state then terminal one will have, which will lead to, obviously, not the best move. This is also an explanation for the constants mentioned before: $0.1, \frac{1}{60}$.

Note: None of the heuristics is proved to be admissible. Moreover, it can be easily seen that they are not admissible.

2.3 Optimization

Algorithm uses $\alpha, \beta-$pruning and this gives significant difference in move times. On Figure 1 there is a plot of move times(y-axis, s) for all moves(x-axis, # of the move) of the game(computer was playing with itself). There were two games played, one had pruning and the other did not.

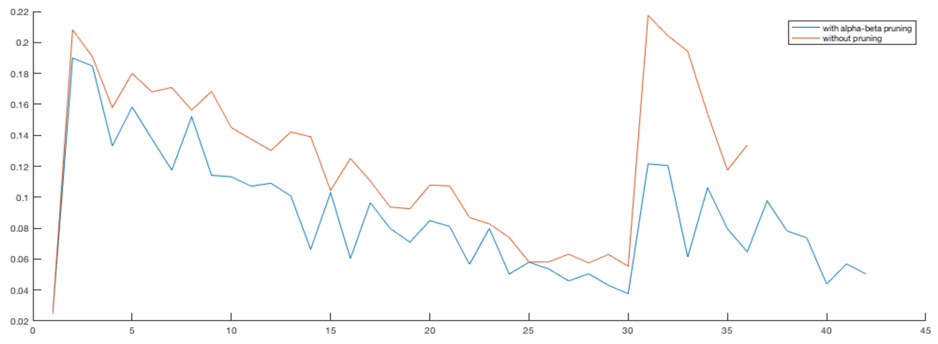

Figure 1: Moves times with pruning and without.

Easy to see, that pruning always gives a positive effect on move times. Up to 30^{th} move difference is not that big as the depth of search is only 3. After it, depth is 4 and amount of computations much larger, so the difference between times is much bigger.

Basically, it's pruning what gives an opportunity to increase the depth of the search by the end of the game.

2.4 Tests and Results

The algorithm was tested by playing with it sometimes, but I'm kind of lazy, so for many games algorithm was playing with http://www.stratigery.com/gen9.html AI for the 9 Board TTT by Bruce Ediger. This AI gives the option to set different strength and I was testing mine with strength 6 and 8. Source says that strength 4 will beat most people. A number of tests I had does not give me right to say that my program beats that one, but I can say that I have seen my program winning. I consider this a good result.

And about testing it with myself: I lose :(That's understandable, as I do not consider myself professional in this game and last time I played it before this assignment, probably, was in 2014. By the way, same time I found this AI by Bruce Ediger and used to play with it.

Another way it was tested is playing with itself. Basically, the game is rather long(often > 40 moves and I even had a 49-move game) and one party wins. That's all that I can say about it. I have not recorded any draws.

2.5 Instructions to run

Just run command "python aosipa_b.py". Everything else will be prompted as needed. Input and output are done according to the requirements in the document. For input sys.stdin is used, other messages are in sys.stderr, computer moves are in sys.stdout.

3 Other variants of TTT

3.1 Super Tic-Tac-Toe

In short: It is not interesting.

This version does not require any AI to play in it as there is a winning strategy for X and therefore this game is deterministic if we will assume that each player does what's best for themselves. Basic TTT can also be considered a quite defined game as state space is not that big, but there, even more, variability than in Super TTT.